Assignment 01 – String Matching

Student Details

- Author's Name – C. A. M. Walopla
- Index No – 20001975
- Registration No – 2020/CS/197

Choose string matching algorithm:-

KMP Algorithm – Knuth Morris Pratt Algorithm

Reason for chosen that algorithm:-

Overview

KMP algorithm is used to check whether the given pattern is included in the text. This algorithm compares characters in left to right and when the mismatch is occurs it uses prefix function to find out earliest next position needed to be checked. According to that value it decide how many characters may have to skip.

- Complexity of KMP algorithm - $O(n+m)$

  n – Length of the text

  m – Length of the pattern

Introduction

KMP algorithm is very algorithm for string searching. When we comparing with the other string matching algorithms like naive string searching algorithm, the time complexity is very high than KMP algorithm. (Naive string matching algorithm time complexity – $O(nm)$, $O(nm) > O(n+m)$) . As a result, the readability, simplicity and understandability is minimize. So, KMP algorithm is the most efficient algorithm for string searching.

Components and Terminology of KMP Algorithm

Suppose the pattern is

Pattern – h o m e

Prefix :

According to the KMP algorithm: there are 2 terms called “proper prefix” and “proper suffix” used for string searching.

- Proper prefix – A subset of the pattern using only beginning portion

  Examples of the prefixes in above pattern are:

  Pattern - h o m e

  Sub string - h ho hom home

  Prefixes - no h h , ho h, ho, hom

Suffix :

- Proper suffix – A subset of the pattern using only last portion

  Examples of the prefixes in above pattern are:

  Pattern - h o m e

  Sub string - h ho hom home

  Suffixes - no o m, om e, me, ome

According, there can be same proper prefixes and suffixes. In the above example that kind (same) of proper prefix and suffix not include.

So, in this KMP algorithm, we may have to find longest proper prefix that is also be a proper suffix. For that purpose we create a table called “LPS table. – Longest Prefix Suffix Table”.

LPS table

The value match with each character, E.g.:- LPS[i] is represent the length of the longest proper prefix and suffix. (0 < i < [length of the pattern -1])

Considering below pattern which include proper prefix and suffix.

Compute LPS table – Method 1

Pattern -

a	b	a	b	a	c	a

Pattern - a b a b a c a

Sub string - a ab aba abab ababa ababac ababaca

Prefixes - no a a, ab a, ab, aba a, ab, aba, abab a, ab, aba, abab, ababa a, ab, aba, abab, ababa, ababac

Suffixes - no b a,ba b, ab, bab a, ba, aba, baba c, ac, bac, abac, babac a, ca, aca, baca, abaca, babaca

Matching length - 0 0 1 2 3 0 1

LPS

0	0	1	2	3	0	1

Compute LPS table – Method 2

Pattern	a	b	a	b	a	c	a
index	0	1	2	3	4	5	6
i value	0	1	2	3	4	5	6
j value	0	0	0	1	2	3	0
Check Pattern[i] ==pattern[j]	Initial point has no prefix or suffix therefore eventually LPS = 0	a != b	a == a	b == b	a == a	b != c	a == a
LPS	0	0	0+1 = 1	1+1= 2	2+1 = 3	0	0+1 = 1

Code for the KMP string matching algorithm

```
# Python program for KMP string searching algorithm
def kmpSearcher(p, t):
    m = len(p)  # getting the length of the pattern
    n = len(t)  # getting the length of the text

    # create LongestPrefixSuffix[] that will hold the longest prefix and suffix
values of the pattern
    LongestPrefixSuffix = [0]*m
    indexP = 0  # index for p[]
    sub_count = 0  # calculating numbers of pattern

    # Preprocess the pattern (calculate LongestPrefixSuffix[] array)
    FindLPS(p, m, LongestPrefixSuffix)

    indexT = 0  # index for t[]
    while indexT < n:
        if p[indexP] == t[indexT]:
            indexT += 1
            indexP += 1

        if indexP == m:
            sub_count += 1
            indexP = LongestPrefixSuffix[indexP-1]

        # mismatch after j matches
        elif indexT < n and p[indexP] != t[indexT]:
            if indexP != 0:
                indexP = LongestPrefixSuffix[indexP-1]
            else:
                indexT += 1
    return sub_count  #returning the how many matches are include in the text
```

Input – Text, Pattern

Output – return the number of matching (pattern is include in the text) count.

Summarization of the algorithm –

- Get the length of pattern and text variables called m and n respectively.
- Declare 2 variables as indexP and indexT which are pointer iterating over the pattern and text.
- Declare a variable called sub_count for counting number of matches include into the relevant text.
- Iterating over the text with check whether the condition (indexT < n) and check, if statement which is p [indexP] == t [indexT] and increment indexP and indexT both of values.
- Also another condition check in this code to find if indexP pointer has reached to the end of the pattern text with match.
- If we found a match then simply increment the sub_count variable value by 1.
- There is a mismatch on the pattern with text and indexP != 0 , we move indexP value into LongestPrefixSuffix [indexP – 1] and if that is not in the case, then increment the indexT value.
- Finally, return the sub_count.

Code for Computing LPS table function

```
# creating a fucntion to compute the LPS table
def FindLPS(p, m, LongestPrefixSuffix):
    len = 0  # length of the previous longest prefix and suffix

    LongestPrefixSuffix[0]  # LongestPrefixSuffix[0] is 0
    i = 1

    # Calculating the LongestPrefixSuffix[i] for i = 1 to m-1 using while loop
    while i < m:
        if p[i] == p[len]:
            len += 1
            LongestPrefixSuffix[i] = len
            i += 1
        else:
            if len != 0:
                len = LongestPrefixSuffix[len-1]
            else:
                LongestPrefixSuffix[i] = 0
                i += 1
```

Input – Pattern, length of the pattern (m), LongestPrefixSuffix that hold the longest prefix and suffix values of the pattern

Output – the LPS table for the relevant pattern.

Summarization of the algorithm –

- Declare a variable as len which store the length of the previous longest proper prefix and suffix.
- Looping over the pattern using pointer variable i until it's value less than length of the pattern (m).
- Value of LongestPrefixSuffix [i] is same as the len variable, then we increment the i value by 1
- But if the character index is not equal to the character in the index len, and also if the len != 0 then the new value of len is become to LongestPrefixSuffix [len-1].
- It the len == 0 , then LongestPrefixSuffix [i] = 0 , and we increment the value of i by 1.

Advantages of the KMP algorithm

- It's time complexity [*O*(n+m)] is less than other string searching algorithms.
- Very fast string searching algorithm compared with any other algorithms.

Disadvantage of the KMP algorithm

- It is very complex to understand.